САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и технологий
Кафедра: "математика и компьютерные науки"

Отчет по задании
по дисциплине"Параллельное программирование".

Работу
Выполнил:
Ли Сыюань
Группа:
3540201/20301

Преподаватель
Лукашин А.А.

Санкт-Петербург
2022

Содержание

Введение

Выбрать задачу и проработать реализацию алгоритма, допускающего распараллеливание на несколько потоков / процессов.

Разработать тесты для проверки корректности алгоритма (входные данные, выходные данные, код для сравнения результатов). Для подготовки наборов тестов можно использовать математические пакеты, например, matlab (есть в классе СКЦ и на самом СКЦ).

Реализовать алгоритмы с использованием выбранных технологий.

Провести исследование эффекта от использования многоядерности / многопоточности / многопроцессности на СКЦ, варьируя узлы от 1 до 4 (для MPI) и варьируя количество процессов / потоков.

Подготовить отчет в электронном виде.

Технология:

C & Linux pthreads

C & MPI

Python & MPI

C & OpenMP

Моя выбранная задача - Алгоритм имитации отжига методом Монте Карло.

Математическое описание

Алгоритм основывается на имитации физического процесса, который происходит при кристаллизации вещества, в том числе при отжиге металлов. Предполагается, что атомы вещества уже почти выстроены в кристаллическую решётку, но ещё допустимы переходы отдельных атомов из одной ячейки в другую. Активность атомов тем больше, чем выше температура, которую постепенно понижают, что приводит к тому, что вероятность переходов в состояния с большей энергией уменьшается. Устойчивая кристаллическая решётка соответствует минимуму энергии атомов, поэтому атом либо переходит в состояние с меньшим уровнем энергии, либо остаётся на месте. (Этот алгоритм также называется алгоритмом Н. Метрополиса, по имени его автора).
Моделирование похожего процесса используется для решения задачи глобальной оптимизации, состоящей в нахождении такой точки или множества точек, на которых достигается минимум некоторой целевой функции F(x) где xϵX (x — "состояние системы",X — множество всех состояний).
Алгоритм поиска минимума методом имитации отжига предполагает свободное задание:
x_0 $\in$X — начального состояния системы;

оператора $A(x, i) : X \times \mathbb{N} \to X$, случайно генерирующего новое состояние системы после i-ого шага с учётом текущего состояния x(этот оператор, с одной стороны, должен обеспечивать достаточно свободное случайное блуждание по пространству X, а с другой — работать в некоторой степени целенаправленно, обеспечивая быстроту поиска);

$T_i > 0$ — убывающей к нулю положительной последовательности, которая задаёт аналог понижающейся температуры в кристалле. Скорость остывания (закон убывания) также может задаваться (и варьироваться) произвольно, что придаёт алгоритму значительной гибкости.
Алгоритм генерирует процесс случайного блуждания по пространству состояний X.Решение ищется последовательным вычислением точек $x_0,\ x_1,\ x_2,\ \cdots$ пространства X; каждая точка, начиная с x_1, «претендует» на то, чтобы лучше предыдущих приближать решение. На каждом шаге алгоритм (который описан ниже) вычисляет новую точку и понижает значение величины (изначально положительной), понимаемой как «температура».
Последовательность этих точек (состояний) получается следующим образом. К точке x_i применяется оператор A,в результате чего получается точка-кандидат $x_i^* = A(x_i, i)$, для которой вычисляется соответствующее изменение "энергии" $\Delta F_i = F(x_i^*) - F(x_i)$. Если энергия понижается $(\Delta F_i \leq 0)$, осуществляется переход системы в новое состояние: $x_{i+1} = x_i^*$. Если энергия повышается $(\Delta F_i > 0)$, переход в новое состояние может осуществиться лишь с некоторой вероятностью, зависящей от величины повышения энергии и текущей температуры, в соответствии с законом распределения Гиббса:
$P(x_i^* \to x_{i+1} \mid x_i) = \exp(-\Delta F_i / T_i)$

Если переход не произошёл, состояние системы остаётся прежним: $x_{i+1} = x_i$. Алгоритм останавливается по достижении точки, которая оказывается при температуре ноль.

Особенности реализации

В соответствии с приведенным выше описанием алгоритм имитации отжига является основным кодом, а язык C используется в качестве примера ниже:

```
void *SA(int *var){
    double x = 0;
    double difference = func(x);
    while (T > eps){
        double dx = -1;
        while (dx < 0) dx = x + ((double)(rand()%(*var)) * 2 - (*var)) * T;//
NOLINT(cert-msc30-c, cert-msc50-cpp)
        double df = func(dx);
        if(df <difference){
            x = dx;
            difference = df;
        } else if(exp((difference - df) / T) * (*var) > (double)(rand()%(*var))){
            x = dx;
            difference = df;
        }
        T *= dT;
    }
    //output information about thread
    printf("Current thread:%d\n",pthread_self());
    printf("current \"x\" equals:%lf\n",x);
    pthread_mutex_lock(&mutex);
        currentErr = fabs(x- sqrt(n));
    if (err == 0) {
        err = currentErr;
        bestResult = x;
    } else{
        if(currentErr < err){
            err = currentErr;
            bestResult = x;
        }
    }
    sum += x;
    pthread_mutex_unlock(&mutex);
    printf("Current sum:%lf\n",sum);
    printf("Final result:%.8f\n\n",x);
    return NULL;
```

}

Суперкомпьютерные запуски

Как показано на рисунке 3, суперкомпьютер подключается через закрытый ключ:

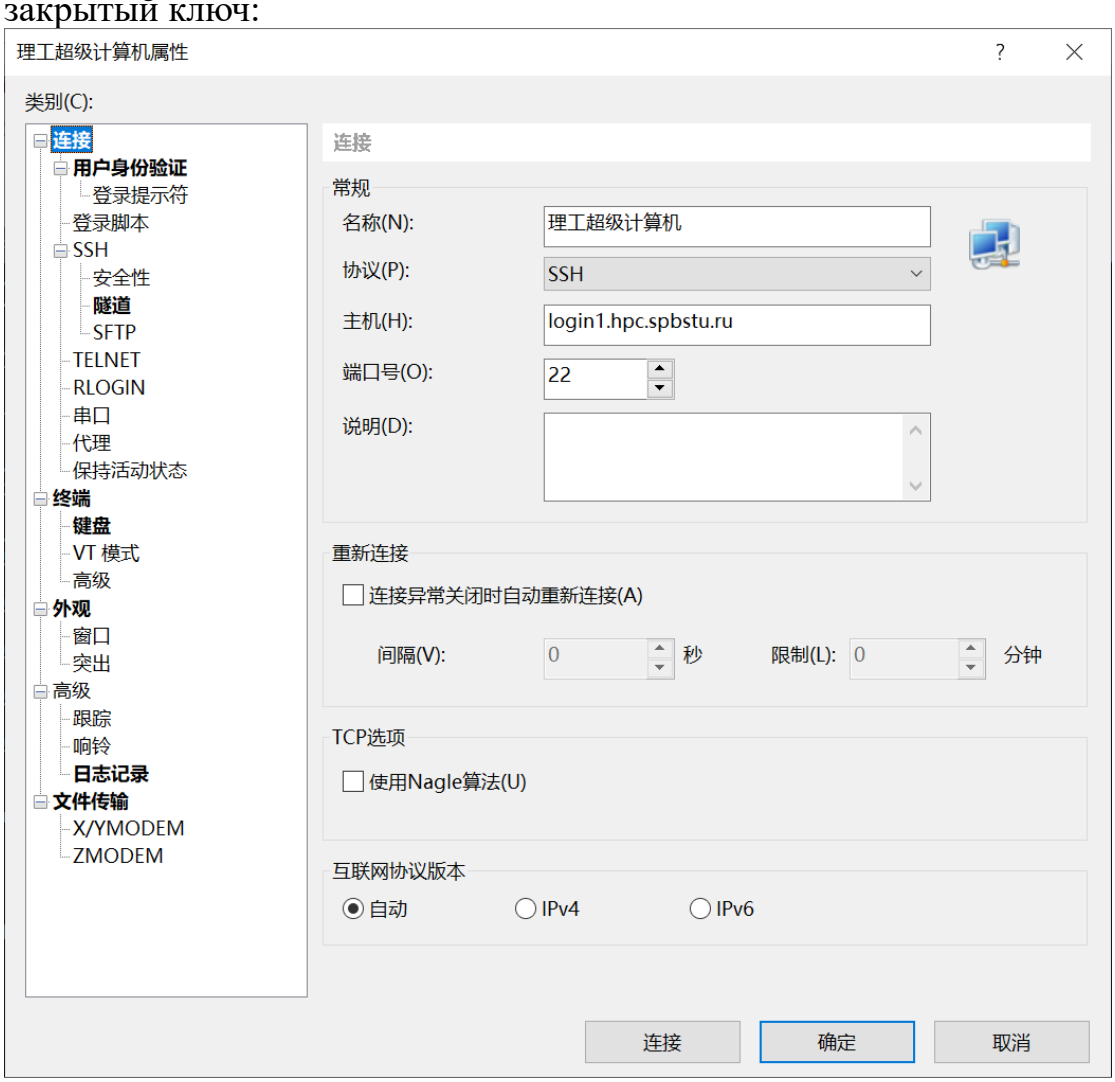

Рисунок 1 – Связь суперкомпьютер

Как показано на рисунке 1, суперкомпьютер успешно подключен:

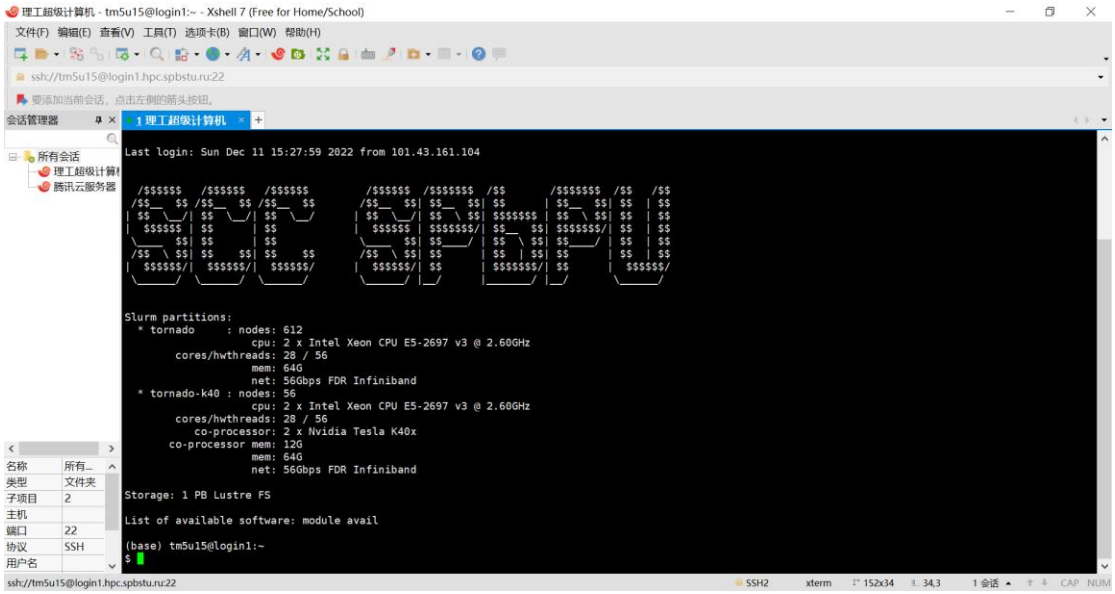

Рисунок 2 – В суперкомпьютеры

Результат работы

По C+pthread

Рассчитайте приблизительное значение корня 4 с Алгоритм имитации отжига методом Монте Карло.

Скомпилируйте и запустите, используя следующие шаги：

```
$ gcc -o main main.c -lm -lpthread
./main
```

При использовании 20 потоков это занимает 1.280998 секунд, а ошибка составляет 0.000075.

```
use time:1.280998
The err is:0.000075
The best result:1.999925
```

Рис3.При 20 поток

При использовании 30 потоков это занимает 1.360381 секунд, а ошибка составляет 0.000005.

```
use time:1.360381
The err is:0.000005
The best result:1.999995
```

Рис4.При 30 поток

При использовании 40 потоков это занимает 1.420276 секунд, а ошибка составляет 0.000010.

```
use time:1.420276
The err is:0.000010
The best result:2.000010
```

Рис5.При 40 поток

При использовании 50 потоков это занимает 1.041333 секунд, а ошибка составляет 0.000010.

```
use time:1.041333
The err is:0.000010
The best result:2.000010
```

Рис6.При 50 поток

По C+MPI

Скомпилируйте и запустите, используя следующие шаги：

```
mpicc main.c -o main -lm
mpirun -np 20 ./main
```

При использовании 20 потоков это занимает 0.090772 секунд, а ошибка составляет 0.000099.

```
Best result:10.000099
err is:0.000099
use time:0.090772
```

Рис7.При 20 поток

При использовании 30 потоков это занимает 0.106633 секунд, а ошибка составляет 0.106633.

```
Best result:10.000099
err is:0.000099
use time:0.106633
```

Рис8.При 30 поток

При использовании 40 потоков это занимает 0.169003 секунд, а ошибка составляет 0.000099.

```
Best result:10.000099
err is:0.000099
use time:0.169003
```

Рис9.При 40 поток

При использовании 50 потоков это занимает 0.279032 секунд, а ошибка составляет 0.000099.

```
Best result:10.000099
err is:0.000099
use time:0.279032
```

Рис10.При 50 поток

По Python+MPI

Скомпилируйте и запустите, используя следующие шаги：

```
mpiexec -n 20 python main.py
```

При использовании 20 потоков это занимает 0.912470 секунд, а ошибка составляет 0.000384.

```
Use time: 0.9124703407287598
err is:0.000384
Final result: 2.000383819849173
```

Рис11.При 20 поток

При использовании 30 потоков это занимает 0.925046 секунд, а ошибка составляет 0.000010.

```
Use time: 0.9250462055206299
err is:0.000010
Final result: 1.9999903003087285
```

Рис12.При 30 поток

При использовании 40 потоков это занимает 1.082019 секунд, а ошибка составляет 0.000020.

```
Use time: 1.0820198059082031
err is:0.000020
Final result: 1.9999802634035886
```

Рис13.При 40 поток

При использовании 50 потоков это занимает 1.220132 секунд, а ошибка составляет 0.000007.

```
Use time: 1.2201321125030518
err is:0.000007
Final result: 1.9999931337868118
```

Рис14.При 50 поток

По C+OpenMP

Скомпилируйте и запустите, используя следующие шаги：

```
gcc -lm -fopenmp main.c
./a.out
```

При использовании 20 потоков это занимает 0.180810 секунд, а ошибка составляет 0.000048.

```
use time:0.180810
Best result is:10.000048
The err is:0.000048
```

Рис15.При 20 поток

При использовании 30 потоков это занимает 0.181597 секунд, а ошибка составляет 0.000109.

```
use time:0.181597
Best result is:10.000109
The err is:0.000109
```

Рис16.При 30 потока

При использовании 40 потоков это занимает 0.200933 секунд, а ошибка составляет 0.000006.

```
use time:0.200933
Best result is:10.000006
The err is:0.000006
```

Рис17.При 40 потока

При использовании 50 потоков это занимает 0.216093 секунд, а ошибка составляет 0.000007.

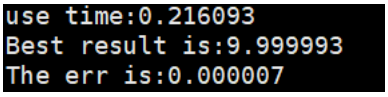

Рис18.При 50 потоков

Заключение

Каждая программа использует для работы 20, 30, 40 и 50 потоков соответственно.Данные в таблице - это расчетная ошибка.Из данных в таблице видно, что общая ошибка постепенно уменьшается с увеличением количества потоков .

Сравнивая время выполнения и ошибку результата, можно увидеть, что результат C+OpenMP самый лучший, время самое короткое, а ошибка самая маленькая.

Поток\Язык	C+Pthread error	C+MPI error	C+OpenMp error	Python+MPI error
20	0.000075	0.000099	0.000048	0.000384
30	0.000005	0.000099	0.000109	0.000010
40	0.000010	0.000099	0.000006	0.000020
50	0.000010	0.000099	0.000007	0.000007
Поток\Язык	C+Pthread runtime	C+MPI runtime	C+OpenMp runtime	Python+MPI runtime
20	1.280998	0.090772	0.180810	0.912470
30	1.360381	0.106633	0.181597	0.025046
40	1.420276	0.169003	0.200933	1.082019
50	1.041333	0.279032	0.216093	1.220132

Приложения

C+pthread

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "pthread.h"
#include<sys/time.h>

//n,求解根号下 n 的近似值
//如果 x 平方减 n 的绝对值越小，说明 x 越接近根号 n 的真实值
```

```
int *ptr;
int thread_count = 0;
//n:calculate number
double n;
//func is used to calculate approximate number,x is the approximate number
double func(double x){
	return fabs(x*x-n);
}
//T is the primitive temperature,every time change:T * dT
double T = 20000;
double dT = 0.9999;
//eps is the ending temperature
const double eps = 0.001;
//Simulated annealing algorithm
double sum = 0;
//err
double err = 0;
double bestResult = 0;
double currentErr = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *SA(int *var){
	double x = 0;
	double difference = func(x);
	while (T > eps){
		double dx = -1;
		while (dx < 0) dx = x + ((double)(rand()%(*var)) * 2 - (*var)) * T;//
NOLINT(cert-msc30-c, cert-msc50-cpp)
		double df = func(dx);
		if(df <difference){
			x = dx;
			difference = df;
		} else if(exp((difference - df) / T) * (*var) > (double)(rand()%(*var))){
			x = dx;
			difference = df;
		}
		T *= dT;
	}
	//output information about thread
	printf("Current thread:%d\n",pthread_self());
	printf("current \"x\" equals:%lf\n",x);
	pthread_mutex_lock(&mutex);
		currentErr = fabs(x- sqrt(n));
```

```
        if (err == 0) {
                err = currentErr;
                bestResult = x;
        } else{
                if(currentErr < err){
                        err = currentErr;
                        bestResult = x;
                }
        }
        sum += x;
        pthread_mutex_unlock(&mutex);
        printf("Current sum:%lf\n",sum);
        printf("Final result:%.8f\n\n",x);
        return NULL;
}

int main(int argc,char* argv[]) {
        struct timeval tv_begin,tv_end;
        gettimeofday(&tv_begin,NULL);
        printf("Please input n:");
        scanf("%lf",&n);
        long thread;
        pthread_t *thread_handles;
        //get the number from the linux commander line
//        thread_count = atol(argv[1]);
        thread_count = 20;
        thread_handles = (pthread_t*)malloc(thread_count * sizeof(pthread_t));
        int sACalRange = 5;
        ptr = &sACalRange;
        //define struct for rand function change range
        for(thread = 0; thread < thread_count; thread++){
                //pthread_create
                pthread_create(&thread_handles[thread], NULL, (void*)SA, ptr);
//                sACalRange += 5;
        }
//    pthread_join，the main thread will wait until the other threads done
        for(thread = 0; thread < thread_count; thread++){
                pthread_join(thread_handles[thread], NULL);
        }
        free(thread_handles);
        gettimeofday(&tv_end,NULL);
        double average = sum/thread_count;
//        printf("average(final result):%lf\n",average);
```

```
	double
time=tv_end.tv_sec-tv_begin.tv_sec+(double)(tv_end.tv_usec-tv_begin.tv_usec)/1
000000;
	printf("use time:%lf\n",time);
	printf("The err is:%lf\n",err);
	printf("The best result:%lf\n",bestResult);

	return 0;
}
```

C+MPI

```
#define _CRT_SECURE_NO_WARNINGS
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//
int rank;
//
int size;
//
double bestResult;
//
double finalResult;
int thread_count = 0;
//n:calculate number
double n = 100;
//func is used to calculate approximate number,x is the approximate number
double func(double x) {
	return fabs(x * x - n);
}//T is the primitive temperature,every time change:T * dT
double T = 20000;
double dT = 0.9999;
//eps is the ending temperature
const double eps = 0.001;
double err = 10000;
double currentErr = 0;
//Simulated annealing algorithm
double SA(int var) {
	double x = 0;
	double difference = func(x);
	while (T > eps) {
```

```
            double dx = -1;
            while (dx < 0) dx = x + ((double)(rand() % (var)) * 2 - (var)) * T;
            double df = func(dx);
            if (df < difference) {
                x = dx;
                difference = df;
            }
            else if (exp((difference - df) / T) * (var) > rand() % (var)) {
                x = dx;
                difference = df;
            }
            T *= dT;
        }
        //
        if (x * x - n < fabs(bestResult * bestResult - n)) {
            bestResult = x;
        }
        //output information about thread
        printf("Current thread:%d\n", rank);
        printf("\"x\" equals:%lf\n", x);
        return bestResult;
}

//
double aggregate_all_results(int my_rank, int comm_sz) {
        double newDiff = 100;
        double diff = 0;
        // Use process 0 to receive all local results and add them.
        if (my_rank == 0) {
            int i;
            for (i = 1; i < comm_sz; i++) {
                MPI_Recv(&bestResult, 1, MPI_DOUBLE, i, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                diff = func(bestResult);
                if (diff < newDiff) {
                    newDiff = diff;
                    finalResult = bestResult;
                    printf("Best Result is Updated,the data is:%lf\n",
finalResult);
                }
                    currentErr = fabs(finalResult- sqrt(n));
            printf("Best result:%lf\n",finalResult);
                    printf("err is:%lf\n",currentErr);
            }
```

```
    }
    else {
        MPI_Send(&bestResult, 1, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
    }
    return finalResult;
}

int main() {
    //

    struct timeval tv_begin,tv_end;
    gettimeofday(&tv_begin,NULL);


    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    SA(5 + 5 * (rank + 1));

    //
    MPI_Barrier(MPI_COMM_WORLD);
    aggregate_all_results(rank, size);

    if (rank == 0) {
        gettimeofday(&tv_end,NULL);
    double
time=tv_end.tv_sec-tv_begin.tv_sec+(double)(tv_end.tv_usec-tv_begin.tv_usec)/1
000000;
    printf("use time:%lf\n",time);
    }
    MPI_Finalize();
    return 0;
}
```

Python+MPI

```
# encoding:utf-8

# 这是一个示例 Python 脚本。

# 按 Shift+F10 执行或将其替换为您的代码。
# 按 双击 Shift 在所有地方搜索类、文件、工具窗口、操作和设置。
```

```
# 在下面的代码行中使用断点来调试脚本。

import random
import time
from mpi4py import MPI
# //当前线程号
from math import fabs
from math import exp
from math import sqrt

rank = 0;
# //全部线程数量
size = 0;
# //当前循环中最佳结果
bestResult = 0;
# //最后结果
finalResult = 0;
thread_count = 0;
# //n:calculate number
n = 4;
# //T is the primitive temperature,every time change:T * dT
T = 20000;
dT = 0.9999;
# //eps is the ending temperature
eps = 0.001;
comm = MPI.COMM_WORLD;


def func(x):
    return fabs(x * x - n);


def SA(var):
    global T, dT, eps;
    x = 0;
    difference = func(x);
    while T > eps:
        dx = -1;
        while dx < 0:
            dx = x + ((random.randint(0, 30000) % var) * 2 - var) * T;
        df = func(dx);
        if df < difference:
            x = dx;
            difference = df;
```

```
            elif exp((difference - df) / T) * var > random.randint(0, 30000) % var:
                x = dx;
                difference = df;
            T *= dT;

        # // 差距如果比当前最佳结果小，则取代
        global bestResult;
        if x * x - n < fabs(bestResult * bestResult - n):
            bestResult = x;

        # // output information about thread
        print("Current thread", rank);
        print("\"x\"approximately equals:", bestResult)
        return bestResult;


def aggregate_all_results(my_rank, comm_sz):
        global bestResult, finalResult, comm;
        newDiff = 100;
        diff = 0;
        # // Use process 0 to receive all local results and add them.
        if my_rank == 0:
            for i in range(1, comm_sz):
                bestResult = comm.recv(source=i);
                # MPI_Recv(&bestResult, 1, MPI_DOUBLE, i, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                print("线程", i, "最好的结果是", bestResult)
                diff = func(bestResult);
                if diff < newDiff:
                    newDiff = diff;
                    finalResult = bestResult;
                    print("Best Result is Updated,the data is:", finalResult)
            endTime = time.time()
            print("Use time:",endTime - startTime)
            err = fabs(finalResult-sqrt(n))
            print("err is:%.6f"%err)
        else:
            comm.send(bestResult, 0);
            # MPI_Send(&bestResult, 1, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
        return finalResult;

if __name__ == '__main__':
        t=time.time()
```

```
startTime = time.time()
print("startTime",startTime)
# MPI_Init(NULL, NULL);
rank = comm.rank;
size = comm.size;
# MPI_Comm_rank(MPI_COMM_WORLD, & rank);
# MPI_Comm_size(MPI_COMM_WORLD, & size);
SA(5 + 5 * (rank + 1));
# //  将各个线程的结果汇总
# MPI_Barrier(MPI_COMM_WORLD);
comm.barrier();
finalResult = aggregate_all_results(rank, size);
print("Final result:",finalResult);
# MPI_Finalize();
```

C+OpenMP

```
#include <stdio.h>
#include "omp.h"
#include "math.h"
#include "stdlib.h"

//当前线程号
int rank;
//全部线程数量
int size;
//当前循环中最佳结果
double bestResult;
//最后结果
double finalResult;
int thread_count = 0;
//n:calculate number
double n = 100;
//func is used to calculate approximate number,x is the approximate number
double func(double x) {
    return fabs(x * x - n);
}//T is the primitive temperature,every time change:T * dT
double T = 20000;
double dT = 0.9999;
//eps is the ending temperature
const double eps = 0.001;
//Simulated annealing algorithm
```

```
//新的最小误差
double newDiff = 10000;
//当前最小误差
double diff = 0;
//该函数用来将各个进程得到的结果汇总起来，以得到最终的近似解
omp_lock_t writelock;
double aggregate_all_results(double bestResult) {
	// Use process 0 to receive all local results and add them.
	//获取当前最好结果
	diff = func(bestResult);
	if (diff < newDiff) {
		newDiff = diff;
		finalResult = bestResult;
		printf("Best Result is Updated,the data is:%lf\n", finalResult);
	}
	return finalResult;
}

// 模拟退火算法
double SA(int var) {
	double x = 0;
	double difference = func(x);
	while (T > eps) {
		double dx = -1;
		while (dx < 0) dx = x + ((double)(rand() % (var)) * 2 - (var)) * T;
		double df = func(dx);
		if (df < difference) {
			x = dx;
			difference = df;
		}
		else if (exp((difference - df) / T) * (var) > rand() % (var)) {
			x = dx;
			difference = df;
		}
		T *= dT;
	}

	//output information about thread
	rank = omp_get_thread_num();
	printf("Current thread:%d\n", rank);
	printf("\"x\" equals:%lf\n", x);
	//加锁比较是否是最优结果，如果是则替换
	#pragma omp critical
	{
```

```
            aggregate_all_results(x);
        }
        return bestResult;
}

int main()
{
        struct timeval tv_begin,tv_end;
        gettimeofday(&tv_begin,NULL);
omp_set_num_threads(50);
#pragma omp parallel
        {
            rank = omp_get_thread_num();
            SA(5 + 5 * (rank + 1));
        }
        gettimeofday(&tv_end,NULL);
        double
time=tv_end.tv_sec-tv_begin.tv_sec+(double)(tv_end.tv_usec-tv_begin.tv_usec)/1
000000;
        double err = fabs(finalResult-sqrt(n));

        // 将各个线程的结果汇总
        printf("\nuse time:%lf\n",time);
        printf("Best result is:%lf\n",finalResult);
        printf("The err is:%lf\n",err);
        return 0;
}
```